Introduction to Linux

Scott Hazelhurst
University of the Witwatersrand
http://www.bioinf.wits.ac.za/courses/linux/

March 2017

Contents

1 Introduction

Material can be found at `www.bioinf.wits.ac.za/courses/linux`

Operating system function

Acts as a software layer to provide access to the underlying hardware

- Higher-level of abstraction
- Sharing of resources
- Memory
- Files
- Processing power
- Communication
- Protection

Examples

Unix-like

- Unix, AIX, Solaris
- Free BSD
- GNU/Linux – Ubuntu, Chrome, . . .
- MacOS X

Windows

- XP
- Windows 7,8

Lots of others

Historical

- IBM 370; OS/2

Mobile

- OSE Symbian

Safety-critical systems:

- Integrity
- OSEK

2 Linux – first look

In these notes, I will refer to Linux and Unix interchangeably. Of course, this is a gross simplification that will upset lots of people. Most of what is covered here holds for operating systems that are inspired by the original Unix system of AT&T. Apple's Mac OSX and Linux are probably the best known examples. Sometimes Linux is called GNU/Linux (also controversial) because although the underlying operating system is Linux, usability relies on many tools that were developed as part of the GNU project. These Unix-like operating systems may be very different on the inside (written by different people and with different architectures) but they are all similar on the outside.

Why GNU/Linux?

- Good quality
- Free
- Many software packages run on it.
- Most common for bioinformatics (and many other scientific areas)

Distributions

Many distributions (flavours) of Linux

- Different look and feel
- Some of the systems programs and environments different
- Mostly for a user interchangeable

Examples

Ubuntu, RedHat, Fedora, Suse, Debian, Scientific Linux, Centos, Mandrake

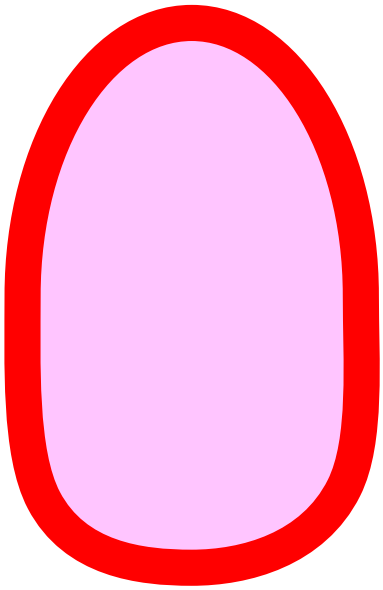
Can download from `www.mirror.ac.za`

Running multiple operating system

- Dual boot
- Multi-boot
- Virtualisation
  - VirtualBox
  - Parallels/VMWare

3 Command line interface

Command line interface/shell



Many different CLI shells – allows interation with OS.

- bash/sh
- csh/tcsh
- many others

On the whole very similar Usually a program like *Terminal* or *xterm* that provides access

- Many shell languages – can be used as a programming language

```
idge 105%
idge 105%
idge 105% cd Teaching/Linux/
idge 106% wc -l overview.tex
    1290 overview.tex
idge 107% /bin/rm *~
idge 108% cp overview.tex  /tmp/2012.tex
idge 109% ls
overview.tex#  exercises.dvi   genintro.aux    misc           overview.nav    overview.tex
inux-GUI.mov   exercises.log   genintro.log    overview.aux   overview.out    overview.toc
uto            exercises.pdf   genintro.pdf    overview.dvi   overview.pdf    overview.vrb
xercises.aux   exercises.tex   genintro.tex    overview.log   overview.snm
idge 110%
```

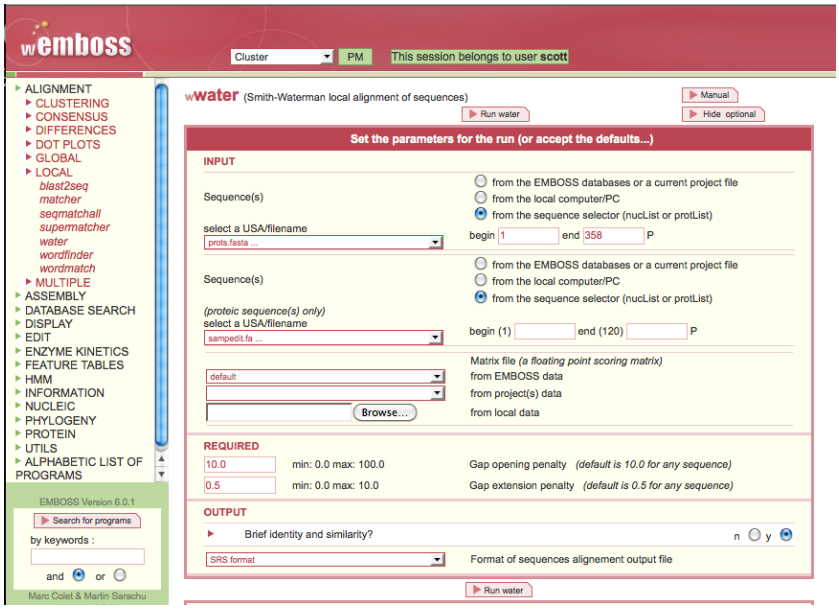
GUI

- Easier to learn
- More intuitive
- Easier mental models
- Memory cues
- Quicker for many things

CLI

- you may not have an option
- More powerful, control
- Great for repetitive tasks

Example 1: run *water*



CLI Equivalent

```
./water -asequence prots.fa  -bsequence sampl.fa \
        -gapopen 10 -gapextend 5 \
        -outfile data.aln
```

Example 2

Move file `Documents/january.txt` to `Data/feb.txt`

- GUI: Click, drag, drop
- `mv Documents/january.txt Data/feb.txt`

Example 3

Suppose you have directories `/opt/data/exp/YY/text/local/control`

- In each directory, there are files `xxxx-YY-month-ddd.xxxx`

Copy the files `xxxx-YY-mar-ddd.eXXX` from all these directories into a directory `/tmp/exp_data/march`

- GUI?
- CLI:

```
cp /opt/data/exp/*/text/local/control/*-mar*.e* \
  /tmp/exp_data/march
```

Example 4

You have new data in a file *myseq.fa* and a directory *db* containing 1875 files.

- Run the *water* program 1875 times to compare your *myseq.fa* file against each of the files in *db* in turn.

3.1 Interacting with Un*x

On modern Unix systems such as GNU/Linux there is reasonably friendly graphical interface. On your screen you are likely to have a windows open. Example windows might be: a web browser, an editor, some applications, and most importantly a shell.

The shell is a window in which you can enter commands to the Unix system – the shell is the interface between you and the system. You may have a number of shells running at the same time.

It is easy to use the shell: just enter the command you want and (a) the system does what you want, or (b) it doesn't and complains that it doesn't understand you.

The conventional syntax of commands issued by the user to a shell is:

```
command command_options command_parameters
```

The command options and ordering of command parameters may differ slightly from version to version of Unix so it is best to use the on-line help provided by Unix to determine the exact available options and parameter order.

A small note on syntax convention. In Unix, the fullstop character does not have any significant purpose within file names and may appear several times in a single file name (and need not occur at all). However,

- To make life easier for ourselves we adopt a conventions of naming files – do `.py` is used for Python programs, `.c` is used for C programs, `.pdf` is used for PDF files, `.tex` is used for LaTeX files, and so on. Binary executables often do not have a suffix.
- Many program use these conventions to guess the contents of files. So a program like Firefox may use the suffix to guess which external program to use to display a file. The LaTeX program expects its main input files to have `.tex` suffixes. But these are conventions that are not enforced by the operating system. Using sensible conventions is as much for your benefit than for the computer's.
- File names beginning with a fullstop have significance to Unix because they contain environment and configuration information useful to both the system and user. Normally files that are start with a "." are not shown when you list the directory and some times not in the GUI file browser.

Command options are also preceded by minus signs to distinguish them from command parameters. For example, `wc data.txt` says count the number of characters, words and lines in the file *data.txt*. But if we include the option `-l` then we only count the number of lines: `wc -l data.txt` If a file name is required as a parameter to a command and is not provided, the shell will by default use the standard input and output i.e. the terminal.

We'll look at what command you can enter and how you interact with the shell in this section.

3.2 Getting started

Initial interaction with the system depends on whether it's the computer in front of you, whether it's on the other side of the world, whether there's a GUI, whether there is security.

Getting started

Normally you log-in

- (Normally), enter user id password to log in
- If running a GUI, run a terminal program

Once terminal runs

- Enter commands
- Case-sensitive
- Kill terminal: Control-D.

Need to log out of session.

Changing password

The standard Un*x command for changing passwords is `passwd` command though there are variants for networked systems.

- Choose sensible passwords

Command history

To see a list of the most recent commands issued to the shell:

- `history`

To repeat a previous command from this list, type:

- `!number`

where number is the number of the command.

Command completion

If you press the tab key, shell tries to complete as much of the command or file name as possible.

- If you type `fi` then TAB, all commands starting with *fi* shown.
- If you type `ls /usr/sh` followed by TAB, system can complete to `/usr/share`
- If you type `ls /usr/l` followed by TAB, system is not able to complete further as there are several options. But if you press TAB twice all the options shown.

Command editing

- The left and right cursor keys, and the delete key allow you to edit the current command;
- Control-A moves to the start of the current command;
- Control-E moves to the end of the current command; and
- The down and up cursor keys allow you to move forward and backward in history.

On-line help

- apropos

  e.g., `apropos music`
- man

  e.g., `man passwd`
- info

  e.g., `info ls`

Exercises

1. Find the File Browser and orient yourself.
2. Identify the following key directories (folders)
   - `/usr` – key system files
   - `/home` – user directories
3. Open the `/usr` directory and look inside the `/usr/bin` directory. This is where most of the programs on the system are kept.
4. Open the `/home` directory. This is where all the user directories are found. Inside this you will find a directory that belongs to you (it has your userid as a name).

   This is your *home* directory.
5. Open up your home directory.
6. Close the File Browser.
7. Note that on the left bar are common programs – you can add or delete. Make sure that you have the Terminal and a good editor there.

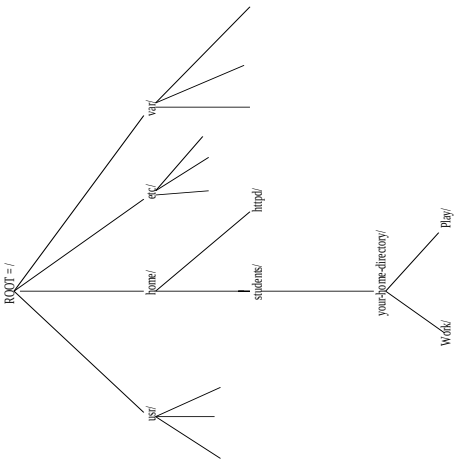


Figure 1: Linux directory Tree

4 Files and directories

All information in Unix is stored in files. The Unix file system is organized so as to allow multiple users to maintain their files in a single hierarchy.

This hierarchy can be viewed as a single tree structure of directories and files with links that may connect to directories and files anywhere else in the structure.

File system
Hierarchical collection of directories and files

Each file has:

1. Contents
2. A Name
3. A location (Directory path)
4. Administrative info. (Who owns it, how big it is, date — see example below)

- The collection of files/directories in a Linux system can be thought of as an inverted tree.
- Top of file system tree is the “Root”, represented by /
- Can have cross-tree *links*

In the GNU/Linux file system tree each user has a unique location to work called their HOME directory. Users are automatically placed in their home directory when they login.

4.1 A word on file and directory names

- A space is a delimiter. Avoid spaces in names.
- forward slash / — separates dir/file names
- backward slash \ treat the next character differently to usual.

```
lpr cats and dogs
lpr cats\ and\ dogs
```

- Avoid: `$`, `~` and quotes in names
- Files starting with `.` are hidden
- Don't start a file name with a –

It's free country: you can flout all these conventions. But don't.

4.2 Manipulating directories

Current working directory

CWD is place in file hierarchy where session is.

- When you first log on you will be in your HOME directory;
- Use the `cd` to change current directory
  `cd dirpath`
- `pwd` command shows CWD.

Many commands by default assume you mean the current working directory.

The system administrator can set up their system so that home directories are where convenient for the organisation

- On Linux systems, often user *bob* is in `/home/bob`
- On MacOS X, `/Users/bob`
- But often variations – safest is to refer to `~bob` or the environment variable `$HOME`

E.g., the `ls` command lists the files in the directory:

- `ls` : lists the file in the current working directory;
- `ls -l` : (small L) lists files in the current directory with a number of details for each file;
- `ls /usr/share` : lists the contents of directory /usr/share.

There are many other options for *ls* – you can use the man page to find out.

Paths

Each file has a *path* – where the file is in the file system.

- Path can be absolute as in `/usr/share/emacs/23.4/etc/JOKES`
  File name is JOKE & path is `/usr/share/emacs/23.4/etc`

Path can be relative to the current working directory

- `JOKES`
  No path given – current working directory is implied
- `funny/JOKES`
  The file JOKES in the directory *funny* that is in the current directory
- `funny/very/JOKES`
  and so on...

Special paths

- `~`
  Tilde: home directory of current user
- `~scott`
  Home directory of user *scott*
- `..`
  Parent directory (relative)
- `.`
  Current directory (relative)

Exercises

All these exercises must be done using the command line interface only. Please do not just to these exercises by rote – you should understand what you are doing and why you are getting the output that you are. There is no prize for finishing the exercises first!

1. Open up a shell using *Terminal*. Note that you can open several at a time. Note that what you do in one shell usually doesn't directly affect any other shells.
2. Check what your working directory is: `pwd`
3. Do `man ls`
4. Do `ls ~`
5. Do `ls -a ~`

4.3 Directory/file information

Directory/file info

Following information kept

- The name
- The name of the owner of the file
- The name of the group of the file
- The date the file was last changed
- The permissions of the file.
- other stuff

The `ls -l` command shows this

4.4 Manipulating files

manipulating files

Examining files

1. `cat`
2. `more`
3. `nano`, `emacs`, `vim`
4. `head -n 25 fname`
5. `tail`

You can use `cat filename` to see the contents of an entire text file on the standard output (terminal).

On the other hand, `more filename` allows you to view the entire text file on the standard output one screen at a time (at the more prompt, a space character will scroll a full screen down while a carriage return character will scroll one line down).

Exercises

1. Using Firefox, download the file `results.csv` from `www.bioinf.wits.ac.za/courses/linux`. Use *more* to inspect.
2. Use *cat* to display the file.
3. Show the first 10 lines of the file.
4. Show the last 25 lines of the file.

4.5 Copying and renaming files

Copying files

To create a copy of a file, use the command:

- `cp  source dest`
- `cp a.txt b.txt`
- `cp a.txt /data/dir`
- `cp -r data backup`

The destination can either be a file (with a path name) or a directory. If it a directory a copy of the file is made and put in the destination giving the copy the original name. If the destination is a file then a copy is made of the source and the copy is given the name of the destination.

Note that by default `cp` does not copy directories. You must use the `-r` option (or another recursive option). Like many commands, `cp` has many options. Doing a `info coreutils 'cp invocation'` will show you these.

Moving a file/Renaming a file:

To move a file from one location to another, use the command:

mv source dest

If the filename in the second parameter is different, a rename is achieved.

Deleting file

`rm`

- `rm data.dat`

4.6 Manipulating directories

Creating, deleting subdirectories

`mkdir`

```
mkdir newdir
mkdir /usr/local/other
mkdir newdir/subdir
mkdir -p newdir/subdir/subsub/other
```

To delete a directory you use the `rmdir` command.

- must be empty

To delete a directory and all its contents, you must first delete the files in the directory using `rm` and then use `rmdir`. An alternative approach is to use `rm -r` for recursive delete. This a very powerful and very dangerous option. There is hardly a system administrator in the world

Changing directory
To change the current working directory, do: `cd newdirpath`

- Change to the directory /usr/share/: `cd /usr/share/`
- Change to user's home directory: `cd ~`
- Change to parent of cwd (current working directory) : `cd ..`
- Change to the directory /etc

  `cd etc`
- Change to the apt directory inside of that `cd ../gnumeric`

Exercises

1. Copy the file `/etc/hosts` to the `/tmp` directory.
2. Create directories `cliex` and `mytest`
3. Create a directories `cliex/one` and `cliex/two`
4. Change directory to `mytest`.
5. Copy the file `/etc/hosts` to the `mytest` directory.
6. Copy the file `/etc/mime.types` to the `mytest` directory.
7. Change your working directory so that you are in the `mytest` directory.
8. Do `more mime.types`
9. Do `tail mime.types`
10. Make directories *january*, *february*, *march*. You only need to use one `mkdir` command.

11. Delete the *march* directory.
12. Change working directory to *january*.
13. Copy the *mime.types* file — now in the parent directory – to the current directory.
14. Copy the *mime.types* file from the parent directory to the *february* directory.
15. Delete all the *mime.types* files you have created.
16. Delete the *mytest* directory and all its contents.
17. Copy all the files in `/usr/share/pixmaps/` to your `pics` directory.
18. List the directory *pics*
19. List all the files in the directory that start with *p*
20. List all the files in the directory that contain *tr* somewhere in the file name.

4.7 Further manipulation of files

The `head` and `tail` commands allow you to see the top and bottom of the file.

- `head snps.txt` show the first 10 lines of the file;
- `head -n20 snps.txt` show the first 20 lines of the file;
- `head -n-3 snps.txt` show everything except the last 3 lines (this doesn't work on standard MacOS X);
- `tail snps.txt` show the last 10 lines of the file;
- `tail -n30 snps.txt` show the last 30 lines of the file;
- `tail -n+3 snps.txt` show everything except the first lines;
- `tail -f snps.txt` show the last 10 lines of the file and then wait for further input. This is useful if you have a program that is writing to a file, and in another shell you want to monitor the output.

Exercise: Try using head and tail on `/etc/passwd`

Extracting colums and rows

Being able to extract out interesting from a file is important.

- cut: extract columns from a file. There are two basic modes (you can read about others in the man file). Columns can be extracted based upon horizontal position (numbering columns by character, using the `-b` option. Or, columns can be extracted by field where the different columns are assumed to separated by field delimiter (by default a tab).

cut – extracting columns

```
cut -b 10-20,30 logs.txt
cut -f 1,2,5,7 results.csv
cut -f 1,2 -d, results.csv
cut -f 1,2 -d\  results.csv
```

The examples above allow us to extact and manipulate files with data. We can do the same thing very simply using a program called Excel and with small files this is probably easier because the GUI allows more intuitive interaction. But consider a realistic example where there are 1000 rows and 10000 columns. Such a large file would be very slow and clumsy to manipulate through Excel or similar program.

The grep command can be used to extract rows based upon what they contain.

grep – extracting lines that match

```
grep rs637812 data.map
grep rs637812 data.map results.map
grep -ir bioinformatics Teaching
```

- Show context

  grep -C 2 rs837812 *map

- Count number of occurrences

  grep -c NA317813 *fam

- Which file

  grep -H NA317813 *fam

- grep -f patterns.txt *

  Use the lines in the files *patterns.txt* as the things to grep for.

Instead of just using plain text files, grep also allows you to search for *regular expressions*, but this is an advanced topic we are not covering now.

There are other powerful tools that can be used too – *awk* and *sed* are very powerful tool that allows extracting and manipulating files.

There are many useful ways of combining files, including *paste* and *join*.

Exercises

For this exercise we are using the *results.csv* file

1. Show everything except the first 5 lines. (This may be useful in many real data files where there may be some headers that you don't need for downstrea analysis)
2. Use *cut* to produce the list of capitals.
3. Use *grep* to find which countries have English as a language.
4. Use *grep* to count how many countries have the West African CFA franc.

4.8 Permissions

File access is determined by the file's protection status. You may control access to your files and directories by granting and denying access privileges to either the user (you), the group the user belongs to (the pg group for example) or all other users. These privileges are read, write and execute.

Permissions

Privileges specified for

- user (owner)
- group
- other

Privileges are:

- read
- write
- execute (files) enter (directory)

Part of a listing of files in a directory may look like this (using `ls -l`):

```
-rw-r--r-- 1 jayesh pg 110 Nov 16 13:11 theory.tex
drwx------ 8 jayesh pg 512 Mar 12 1992  Personal
-rwxrx-r-- 1 jayesh pg 230 Aug 27 12:52 a.out
```

Here we have three files: *theory.tex*, *Personal* and *a.out*. In each case the owner is the user *jayesh* and the group is *pg*.

The 10 characters on the left indicate file protection and is interpreted in the following manner. An `r` indicates that the user can read the file and `w` means the user can change the file. The `x` stands for *execute*. For ordinary files, an `x` indicates that this file stores a program that is ready to run and can be run. For directories, `x` indicates permission to enter the directory.

1. character 1 – if there is a dash the file is an ordinary file; if it is a `d`, this is a directory; if it is a `l` this is a link.
2. Characters 2–4 indicate the permissions of the owner of the file.
3. Characters 5–7 indicate the permissions of members of the group.
4. Characters 8–10 indicate the permissions of other users.

In the example above, the user jayesh can read and change the file *Theory.essay*, members of the *pg* group and indeed all other users can just read it. The file *Personal* is actually a directory. jayesh can read and write to this directory, and also enter the directory. Noone else can access the directory. The file a.out can be read, written and executed by jayesh. Members of the *pg* group can read and execute the file, while all other users can just read it.

Changing permissions

`chmod` changes permissions on any files.

- *chmod g+rwx file1* will give members of the group associated with the file the ability to read, write and execute it.
- Or *chmod o-rx file1* will remove read and execute privileges or other users.

- *chmod ug+x file1* gives the owner (user) and group permission to execute.
- *chmod a+rwx* will give all users permissions to read, write and execute files.
- *chmod o=r file* gives others the ability to read the file and removes any other permissions that others may have.
- *chmod g=rx,o=r file* gives group ability to read and execute, others the ability to read the file and removes any other permissions previously had

Exercises

1. Copy the file results.csv into a file money.txt.
2. Give the results.csv file permissions so that the owner can read it but there are no other permissions.
3. Try to edit the file.
4. Try to delete the file... Why are you actually allowed to delete the file?
5. Give permissions to the money.txt so that the owner can read and write and others can read.

Directories

- For directories, *x* means that the permission holder can *enter* the directory.
- The permission *X* means: if the file is a directory, then *x*, otherwise ignore.

  ```
  chmod a+X *
  ```

  Numeric permission:
- 4: read permission. 2: write permission. 1: execute permission.
- 5=read and execute. 3=write and execute. 7=all
- Typically given as as triple UGO user-group-other:

  752

Permissions can be combined

```
chmod u=rwx,g+w,o-w data
```

This says: set the directory data's permission so that the user can read/write/enter(execute); in addition to any existing permissions allow member's of group of the directory to write to it (i.e., add or remove files from the directory), and remove write privileges from everyone else.

Exercise: Give your own home directory the following permission.

- user: read, write, execute
- group: read, execute
- other: execute

For each directory in your home directory set the permissions so that you can read, write and execute but that no-one else has any permissions. The only exception is `public_html` which you should give read and execute privileges to all users.

There are more complex permissions possible too in the standards permission model together with access control lists. But this is beyond our course.

4.9 Globbing

Globbing

Ways of specifying many things at the same time

- `*`
- `[]`
- `{}`

Wildcard

- `cat *`
- `ls a*.dat`
- `ls *mar*.e*`
- `/bin/rm hello-201[01].[oe]xx`
- `ls *.{tex,aux}`

5 Process Control

Process control

Many jobs can execute at the same time (same, different users)

- foreground
- background

Jobs may belong to currrent shell, or others

- Each job as a PID (global)
- Each job has job number (specific to that shell) Use % to refer to this

Viewing jobs

Jobs in current shell

- `jobs` shows job number
- `ps` shows PID

See other jobs

- `ps -u scott`
- `ps -a`

Manipulating jobs of current shell

If the job is in the foregrond

- Control-C kills
- Control-Z suspends

Can reactivate suspended jobs

- `fg` put it foreground
- `bg` put in background
- `bg %3` (job 3 – if several suspended jobs)

Killing any job

```
kill pid
kill 3617
kill -9 3617
```

Running a job

To run a job in the foreground give the command:

```
emacs
python hello.py
xcal
xeyes
```

To start in the background put an ampersand afterwards

```
xeyes &
emacs newprog.py &
```

Exercises

1. Download the files *spin* and *spin2* files from `www.bioinf.wits.ac.za/courses/linux`
2. Put them into the `~/Linux` directory (create this directiory if needed).
3. Give the *spin* files read and execute permissions for everyone.
4. Look at the *spin* program
5. Run the spin program

   ```
   ./spin
   ```

6. The program is written in the bash language and you should make sense of what is happening.
7. Now run the spin2 program several times

```
./spin2 &
./spin2 &
./spin2 &
./spin2 &
./spin2 &
./spin2 &
```

The program *spin2* does pretty much the same as the *spin* program except it does it silently.

Monitoring Load

- `top`

  The *top* command shows the processes that are using most of the CPU. There's a whole lot of other info you can get too.

- The `w` command shows you who's logged on, as well as current load.

top also shows you memory usage. This can often be critical for system performance. The most important columns in this respect are `%MEM` and `RES`.

Exercises

1. How do you monitor the load given by a specific process?
2. Run the `jobs` command. This shows all these *spin* jobs running together with their job number (not their PID).
3. Kill one of them: `kill %4`

   Note the percentage sign
4. Bring one them to the foreground

   `fg %3`
5. You will now see that the job controls the terminal.
6. Suspend that job: Control-Z
7. Do a `jobs` again.
8. Now do a `ps -u user`

   Note that the *user* is because that is your userid in the lab. On my machine, I would say `ps -u scott`
9. This lists the jobs by process ID.
10. If I run this on my system, I get something like this (among others)

```
501   700 ttys001    0:00.00 /bin/bash ./spin2
501   702 ttys001    0:00.00 /bin/bash ./spin2
501   704 ttys001    0:00.00 /bin/bash ./spin2
```

You'll see different numbers – in this example, the PIDs are 700, 702, 704.

11. Kill the first one you can see.
    `kill 700`
12. Do a `ps` again
13. Do a `jobs`
    Oops, we forgot we suspended one — re-animate it `bg %3`

Aside – discovering resources

- Load needs to be compared to the resources on the system
- /proc/cpuinfo
- /proc/meminfo

6 Remote work and data transfer

Remote work and data transfer

Being able to communicate with other computers is key.

- Each computer has an address
  IPv4: 192.168.2.168
  This allows other computers to communicate with it.
- Not human-friendly: DNS allows us to associate a name
  `grid.core.wits.ac.za` — 146.141.240.100

6.1 wget

wget

The **wget** program can be used to fetch files across the network given a URL:

```
wget www.bioinf.wits.ac.za/courses/linux/seq01.fa
```

wget is particularly useful for doing mass download of files in a directory. You can do recursive download like so:

```
wget -r --no-parent www.bioinf.wits.ac.za/courses/linux/seq01.fa
```

Be careful with this as you can do anti-social and stupid and costly things with this. The `-r` option says do recursive download, the `--no-parent` option says don't follow the `..` directories upwards. Normally wget does traverse up the directory path. This is a slighly silly default since if you forget the `--no-parent` option you can do really silly things. Actually *wget* has some very useful features. Again, look at the man page.

6.2 Logging on to remote machines

ssh

You can use the `ssh` program to do this – of course you must have an account on the remote system

```
ssh gate.place.ac.za
ssh -Y gate.place.ac.za
ssh -Y crick@gate.place.ac.za
```

scp program copies

```
scp file.txt remote.place.wits.ac.za:data
scp -r exps  remote.place.wits.ac.za:data/mon
```

For more detail see `http://kimmo.suominen.com/docs/ssh/`

Passwordless login

Passwords are ubiquitous but have problems

- Annoying to use each time, especially when working remotely
- Often a weak link in system security

ssh supports public/private key authentication

- Can be complemented by passphrases and key-agents

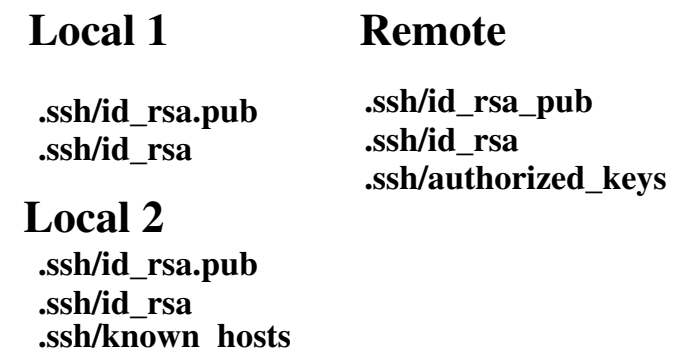
PKI/PKA

In PKA schemes each person (and computer) has

- public key : indentifies you
- private key : (kept very secret) proves/authenticates you

Generate with `ssh-keygen`

Overview



Exercises

1. NB: This assumes you do not already have ssh keys. If you do, please don't overwrite them!
2. Check whether you have existing keys

   ```
   ls -ld ~/.ssh ~/.ssh/*
   ```

3. If not, generate a new key – 2048 bit long RSA

   ```
   ssh-keygen -t rsa -b 2048
   ```

4. You will be asked where to save this. The default is `./ssh/id_rsa` so you can just press enter.
5. Then you are asked for a passphrase – you can just choose an empty passphrase. Press enter twice.
6. Check whether you have new keys

   ```
   ls -ld ~/.ssh ~/.ssh/*
   ```

7. Look at the permissions. SSH is very very sensitive to correct permissions. If you get them wrong ssh won't work.
8. Now the PUBLIC key has to be moved to the remote computer (e.g, physically with a USB drive, email, scp with password).
9. It should be placed in the `~/.ssh/authorized_keys` file. This file can have any number of ssh PUBLIC keys.

6.3 screen – window manager for command-line

screen allows separation of terminal from physical device

- multiple virtual terminals on same device;
- move terminal session from one physical device to another;
- robust against time-outs

Particulary useful for remote access
screen has many features. In this section, I am only showing a selection of the key features, but some very useful features are being omitted.

Creating a new screen

```
screen -S name
```

Creates a new terminal session, putting existing session in background

- terminal session gets a unique number
- named by some user-defined meaningful identifier.

- terminal session destroyed when you do a C-d
- terminal session *detached* with *C-a d*

Listing terminal sessions
The command is `screen -ls`

Note that each terminal has the name you gave it and a unique number.

Reattaching terminal sessions
You can reattach a session to any terminal.

```
screen -r name
```

One slightly annoying feature of *screen* is that it uses C-a as the default escape key sequence (e.g., C-a d detaches the current session). This is annoying because C-a is a commonly used key sequence in editing commands (go to the beginning of the current line). You can chang the behaviour by using the `-e` option. For example,

```
screen -e^Mm -S update
```

will create a new terminal session called *update*, and instead of C-a being the escape sequence, C-m will be.

6.4 Exercises

1. Create a new session with screen: `screen -S A`

   Run a few commands – e.g., ls, cd – just to show you can

   Execute: `echo "Session AAAAAAAAAAA"`

   Now end the terminal session: C-d

   You now go back to your original session and the new screen sesion was destroyed.

2. `screen -ls`

   You should see nothing.

3. Now repeat exercise 1, but instead of ending the terminal session, *detach* it: `C-a d`

4. `screen -ls`

   You should see the session that you have just detached.

5. Now, reattach the session: `screen -r A`

   And once you've convinced yourself that you have your session back, detach it.

6. Start a new session: `screen -S B`, and do `echo "Session B"` so that you can identify it.

   Detach: `C-a d`

   Do: `screen -ls`

7. Start a new session: `screen -S C`, and do `echo "Session C"` so that you can identify it. For the moment don't detach it.

8. Now, open another Terminal using your computer (i.e., not another screen session, but another terminal using computer's windowing system). Do a `screen -ls` Now, you can reattach your session to the current computer: `screen -r B`

   Convince yourself that you have the right terminal session and then detach.

   You can also attach a terminal session that is already active elsewhere:

   `screen -d -R C`

   This detaches session *C* from where it was previously attached, and attaches it to your new terminal.

9. This ability to detach on one terminal and reattach on another is particularly useful if you are logging into the system remotely (using ssh for example). For example, if you are doing administration on your centre's server, you might ssh in from your work desktop, create a session do some work, detach the session. That evening, you can remotely log in from your home computer and then reattach the terminal there.

6.5 Environmental variables

Each shell has a set of *environment variables*

- Used by programs to guide behaviour
- Environment vars set per-terminal – modifying one doesn't modify others
- Init set up in `/etc/bash.bashrc` (system-wide) and `.bashrc` (personal)
- To see environment variable: `printenv` or `printenv VARNAME`
- To use an environment variable – prefix with $, e.g. echo $HOME
- Set environment variables using:

  export PATH=/opt/local/bin:${PATH}

  (note no dollar on left-hand side)

The PATH environmental variable tells the system where to look for executables. These directories are searched in order, and the first path found with the named executable will be used.

Typically the current working directory is *not* on the PATH. That is why when you want to execute a script in the current working directory that you have to say `./myscript` rather than just `myscript` This helps (a little) in preventing inadvertent execution of scripts.

Key environment variables

- PATH: binaries
- PYTHONPATH
- R_LIBS
- PERL5LIB
- PYTHONPATH

- HOME
- HOSTNAME / HOST
- USER

Initialisation

Environment variables (and other initialisation) can be set by start up scripts

- `/etc/profile`
- `.bash_profile` : login shells
- `.bashrc` : other interactive shells

Can use the *source* command to load another initialisation script.

7 Advanced topics

Archiving

Most common archive command is `tar`

Creating an archive

```
tar -cf newarchivename.tar directory

tar -cf exercises.tar /home/scott/progs/python-ex
```

To unarchive
tar -xf exercises.tar

Compressing

`gzip`, `gunzip` are common compress/uncompress pair

- .gz suffix

```
tar -cf ex.tar progs/python-ex
gzip ex.tar
...
tar -xzf ex.tar.gz
```

Other compressing, archiving

bzip2, bunzip2

- Most compressed: bz2 suffix

zip: compatible with other systems, not as efficient

- zip -r myarch.zip existdir

Installing software

Many ways of doing

- untar, run config, make, compile
- use package manager
  - rpm
  - yum
  - apt
  - fink
  - git

8 Pipes and Redirection

8.1 Combining processes

Most of the commands we have seen so far produce output on the terminal; some may also take input from the terminal. Unix allows the terminal to be replaced by a file for either or both of input and output. This becomes useful for interaction required by background processes as well as for allowing users to combine commands to create their own commands.

Input-output redirection and pipes are provided by Unix for this purpose. Input-output redirection The syntax for output and input redirection are given by the following forms respectively:

Redirection

```
command  >filename
command < filename
```

The symbol $>$ means "put the output in the following file, rather than to the terminal" and the symbol $<$ means "get the input to this command from the following file, rather than from the terminal".

>> used instead of > *appends* output to the file rather than overwriting contents of the file.

For example, the command sequence

```
ls > /tmp/wdirfiles
wc -l < /tmp/wdirfiles
```

can be used to count the number of files in the working directory.

There are more advanced features too.

- >> can be used to append to a file.
- A common use of *cat* and redirection is to combine file.

```
cat *.dat > all.dat
cat dates*.csv | sort | uniq > all_dates.txt
```

8.2 Pipes

Pipes

Pipes allow you to glue together programs

- output of one program becomes input of the next

```
ls | wc -l
```

It is common practice to put the output of one program into the input of another via a temporary file as in the example above. This can be achieved by first using a command with an output redirection followed by one with an input redirection. However, by doing so we incur a storage overhead cost for a temporary file to store the input to the second program. Furthermore, it would be more efficient to run both programs in parallel so that a continual output from the first program can be fed into the second program. This observation leads to one of the fundamental contributions of the Unix system, namely the pipe.

A pipe (denoted by a vertical bar i.e.. `|`) is a way to connect the output of one program to the input of another without any temporary file; a pipeline is a connection of two or more programs through pipes. All programs in the pipeline execute in parallel to achieve good performance. Only data dependencies restrict the flow of data between these programs. Any program that reads from the terminal can read from a pipe instead and any program that writes on the terminal can write to a pipe. This is where the convention of reading the standard input when no files are named pays off: any program that adheres to the convention can be used in pipelines. grep and sort are two examples often used in pipelines.

For example, the command: `who | grep mary | wc -l` counts the number of times user mary is logged on.

Example
List in alphabetic order the capitals of the countries which use the West Africa CFA franc.

Get the data file and inspect it

Now extract column data and sort

```
grep "West African CFA" results.csv |
     cut -f 2 | sort
```

8.3 xargs

Converts standard input to arguments.

Suppose we have a file that contains status of various files

```
january.dat    GOOD
feb.dat        BAD
march.dat      BAD
april.dat      GOOD
may.dat        BAD
...
```

How can we delete the bad files?

Find the bad files

- `grep BAD statusfiles.csv`

Extract out the names

- `grep BAD statusfiles.csv | cut -f 2`

Delete them
Extract out the names

- `grep BAD statusfiles.csv | cut -f 2 | xargs /bin/rm`

Programming in bash

Can type commands in shell, or save in file and run

- e.g., doexps.sh
- hash-bang
- make executable
- run like so `./doexps.sh`

Normally, an executable program is expected to be machine code that can run directly on the machine. If the program is a script in bash or some other language, you need to tell the system. One way of doing it is to explicitly tell it:

- bash doexps.sh

But it is useful both to save typing and more importantly to help users of your scripts to have a way of telling the system how to interpret your file. Recall that the name of the file does not have this information. The hash-bang is a special sequence on the first line that specifies how the script should be executed. For example, a bash script would have

- `#! /bin/bash`

and if it were a Python program

- `#! /usr/bin/python`

Example script

```
#! /bin/bash
N=10
BASE="gwas14"
BED=${BASE}.bed
BIM=${BASE}.bim
FAM=${BASE}.fam
plink --bfile sample --bmerge $BED $BIM $fam --make-bed --out xxx
```

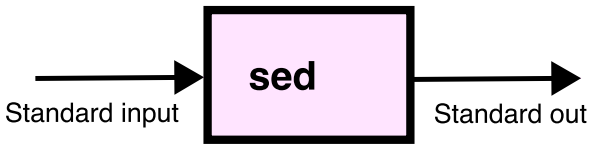
Can pass a parameter

Example script

```
#! /bin/bash
N=10
BASE=$0
BED=${BASE}.bed
BIM=${BASE}.bim
FAM=${BASE}.fam
plink --bfile sample --bmerge $BED $BIM $fam --make-bed --out xxx
```

8.4 Stream editor: sed

sed is a very powerful utility that can be used to transform files or pipes and is very useful for system administrators. We are only going to see a small subset of the features.



As the standard input is *streamed* through `sed` a transformation happens, usually on a line-by-line basis.

Basic operation:

- A condition to match a line
- What transformation must happen

Plus auxiliary switches and options
Example:

```
cat results.csv | sed "s/french/FRENCH/"

sed "s/french/FRENCH/"  < results.csv > new.csv
```

sed / regular expressions

Regular expressions are widely used in system administration, though minor variations in syntax and semantics (very annoying)

- Generalised way of describing a set of patterns
- Alternative to globbing – more powerful
- `man re_format`

Simplified description:

- Most characters stand for themselves: linux, windows, 2015-03-07
- A number of characters that are special

  `^.[$()|*+?{\`

- Can escape to get the actual characters

.	matches anything (except EOL)
[ ]	matches a set of chars
( )	used to group things
—	choice/alternation
*	repeat regex on left 0 or many times
+	repeat regex on left 1 or many times
\	escape character

- *ab.d* : matches abcd, abdd, ab7d, ab:d etc.
- *ab[aeiou]c* : macthes abac, abec, abic, etc.

Loops

```
for x in a b c d; do
  blastx -i $x/data.dat -d see.db -o $x.out
done
```

Backtick
Executes command, and returns the value.

```
date
X="date"
Y=‘date‘
echo X
echo Y
```

Loops

```
for x in ‘seq 100 200‘; do
  echo Backing up $x
  ssh 192.168.30.$x /opt/sysconfig/backup
done
```

Gaining super-powers

Actions allowed determined by permissions

- Some actions only allowed for *root* user
- Some users can be authorised to act for *root*

sudo

```
ls /root

sudo ls /root
```

Becoming someone else

```
su bob
```

Must know *bob*'s password (or be root)

Getting super-powers for a session

```
# if you know root's password
su

# if you are a sudoer -- use your password
sudo su
```

It is strongly recommended to use sudo rather than su. This limits inadvertent damage you can do to your system.

9 Introduction to Maui/Torque

Maui/Torque is the job scheduling system that we run. This system is responsible for allocating jobs to worker nodes. Through the use of the scheduling system you can run many jobs at the same time, and Maui/Torque takes care to allocate the jobs sensibly, rather than your running it manually. Also Maui/Torque shares resources fairly between users.
Used for computer clusters:

- head node (e.g., cream-ce.core.wits.ac.za)
- many worker nodes
- storage other servers

Need to share fairly and effectively – queueing sytem

- User submits job on the head node to the queue

The main commands you need are

- `qsub` for running jobs
- `qstat` for looking at the queue
- `qdel` for killing a job.

To run a job create a text file:

- PBS directives
- Actual commands to execute (as you would type them in)

The basic idea is that you put the job you run into a job script: this is essentially a shell script with some Torque (aka PBS — an earlier version of Torque) directives. These directives describe resources that users need. It's in your interest to be as accurate as possible. The fewer resources your job demands the sooner it will run; however, if your job exceeds its resource demands, your job will be killed off.

PBS Directives
The basic things you can ask for are:

- how many computers (and processes) you want;
- how much memory you want;
- how long you want your job to run.

To repeat:

- Be honest for two reasons:
  - It's the right thing to do and respectful of other users.
  - If you lie or make a mistake (e.g., underestimate the amount of memory you need), your program will suffer. For example, suppose you say you need 8GB when in fact you need 20GB. The worker node will run your program, limiting your space. The likely effect is that your program will thrash and you are going to get CPU utilisation of less than 1%.

Example – job1.sh

```
#PBS -N MyJobName
#PBS -q WitsLong

hostname
date
```

From the command line: `qsub job1.sh`
Output and error files are placed in the directory from which you ran

Example 2 – motion.sh

```
#PBS -N PerpetualMotion
#PBS -q WitsLong
#PBS -l walltime=01:30:00,mem=16GB

wcd -o data1.clt -c sample1.fa
wcd -o data2.clt -c sample2.fa
python ${HOME}/bin/ricompute.py data1.clt ideal1.clt
python ${HOME}/bin/ricompute.py data2.clt ideal2.clt
```

This says run the job for a maximum of 1 hour and 30 minutes on any machine that has at least 16GB of RAM. This has also shows a hypothetical example of a job script – in this case it runs a program called *wcd* on two data sets producing various output, and then a Python program called *ricompute.py* is used to process that output further.

You can put as many commands or instructions in a job script as you wish. Remember that *all* the processing for the job must be under the time limit for the job. Also remember that the commands in the job script are run sequentially: the second *wcd* call only starts after the first completes; the Python program is only called after the *wcd* program completes.

You submit many jobs in parallel. Another way to run the above would be to have two scripts

```
------------  job1.sh
#PBS -N PerpetualMotion-1
#PBS -q WitsLong
#PBS -l walltime=01:30:00,mem=16GB


------------  job2.sh
wcd -o data2.clt -c sample2.fa
python ${HOME}/bin/ricompute.py data2.clt ideal2.clt
```

and then run the jobs in parallel by saying

```
qsub job1.sh
qsub job2.sh
```

Monitoring

When you run qsub, you get a job number back.

```
qstat
qstat -u scott
checkjob 3128822
qdel 3128822
```

10 Unix command reference

This section is best used as a quick command keywords reference. The on-line help facility provided by Unix is a good starting point for finding detailed syntax and semantic information on these commands.

Command	**meaning**
login	sign on
logout	exit
passwd	change login password
Manipulating files and directories	
cat	concatenate and print on standard output
cd	change directory
chmod	change modes or permissions on files.
cp	make copy of files.
find	finds designated files in directory tree and acts upon them.
cut	extract columns from file.
grep	search files for lines matching pattern
head	return top of a file
ln	make file links.
ls	list contents of directory.
mkdir	make a new directory.
more	view long files one screenful at a time.
mv	move or rename files.
rm	remove files.
rmdir	remove directories.
sort	sort lines in file
tail	return bottom of file
Other	
at	execute commands at a later time.
apt-get	(Ubuntu) Install packages
apt-cache	(Ubuntu) Search for packages
date	gives date and time.
diff	finds the difference between two files or directories.
emacs	an editor for creating files
finger	provides information about users.
gcc,g++	Compiler for C, C++
gdb	Debugger for GNU/Linux
history	prints a list of commands issued to the shell.
jobs	list suspended and background jobs.
kill	will terminate jobs.
man	find manual information be keyword.
ps	generate a process status report.
pwd	prints working directory path name.
scp	copies to/from remote machine
screen	CLI windows manager: list `-ls`; reattach: `-r`; create `-S`. Detach: C-a d
sed	stream editor
ssh	allows log-in to remote machine
time	will time how long a command takes.
w	who is on the system and what they are doing.
wc	word count.

Note: This command reference is by no means a complete list of commands available in Unix but contains some commonly used commands.

Other references

http://tldp.org/LDP/intro-linux/html/

Index